



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Scalable Accelerator for Nonuniform Multi-Word Log-Quantized Neural Networks

Jooyeon Choi

Department of Electrical Engineering

Graduate School of UNIST

Scalable Accelerator for Nonuniform Multi-Word Log-Quantized Neural Networks

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Jooyeon Choi

06. 10. 2020

Approved by



Advisor

Jongeun Lee

Scalable Accelerator for Nonuniform Multi-Word Log-Quantized Neural Networks

Jooyeon Choi

This certifies that the thesis/dissertation of Jooyeon Choi is approved.

06. 10. 2020

signature



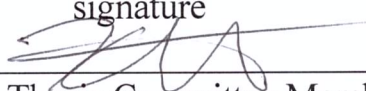
Advisor: Jongeun Lee

signature



Myeongjae Jeon: Thesis Committee Member #1

signature



Kyuho Lee: Thesis Committee Member #2

Abstract

Logarithmic quantization has many hardware-friendly features, but its lower accuracy in certain conditions has prevented more widespread use. Recently modified schemes have been proposed to solve the accuracy problem without compromising its hardware efficiency by selectively employing multiple words. This however causes variable-latency multiplication, demanding a new hardware architecture to support efficient mapping of large neural network layers as well as various types of convolution layers such as depthwise separable convolution. In this paper we present a novel hardware architecture for nonuniform multi-word log-quantized neural networks that is scalable with the number of processing elements while maximizing data reuse. Our architecture supports depthwise convolution and pointwise convolution as well as 3D convolution, which are important for recent mobile-friendly networks. We also propose a hardware-software cooperative optimization to reduce the impact of variable-latency multiplication on performance. Our experimental results using various convolution layers from MobileNetV2 demonstrate the speed advantage of our architecture and high scalability with the number of PEs, compared with previous architectures for depthwise separable convolution or log quantization. Our results also show that our optimization is very effective in improving the performance of our architecture.

Table of Contents

I . Introduction	-----1
II . Background and Related Work	-----3
2.A. Linear vs Logarithmic Quantization	-----3
2.B. DNN Hardware Accelerators	-----3
III . Our Proposed Architecture	-----5
3.A. Hardware Dataflow	-----5
3.B. Consideration for Other Types of Convolution	-----8
3.C. Main Datapath	-----9
3.D. Mapping Examples	-----10
IV. Maximizing Input and Output Data Reuse	-----12
4.A. Architecture Overview and OFM Data Path	-----12
4.B. Weight Data Path	-----13
4.C. IFM Data Path	-----13
V. Handling Variable Latency Operations	-----15
5.A. DWC	-----15
5.B. PWC and Column Reordering	-----15
5.C. Column Reordering Algorithm	-----16
VI. Experiments	-----18
6.A. Experimental Setup	-----18
6.B. Hardware Synthesis Result	-----18
6.C. Performance and Effect of Column Reordering	-----20
6.D. Effect of the Number of PE Planes	-----21

6.E. Training Result	22
6.F. Scalability	23
6.G. Comparison with Previous Work	23
V. Conclusion	24

I. Introduction

Logarithmic quantization, or log quantization for short, encodes numbers in the logarithmic scale, meaning that log-quantized numbers are more densely populated in smaller-valued regions as opposed to larger-valued regions. This property of log quantization turns out to be very well-suited for encoding weight parameters of neural networks. LogNet[1], for instance, demonstrated that 5-bit log-quantized weights can outperform 7-bit linear-quantized weight in the case of AlexNet. Moreover, log quantization is more hardware friendly. Log-quantized weights are shorter in width, hence takes less space in memory and less power to access. Using log quantization, multiplication can be reduced to either addition or shift, which not only makes hardware much faster but also reduces area and power considerably.

However, log quantization, in its original form, performs very poorly for larger neural networks, or those that require higher precision arithmetic. This is true even if we increase the resolution of quantization (i.e., the number of bits used for encoding data) indefinitely [2]; increasing resolution for log quantization has nearly no effect on average precision after a certain point.

To mitigate the limited precision problem of log quantization, multiple techniques have been proposed recently [2], [3], the common idea being that precision can be improved significantly by employing more than one log-quantized numbers.

By representing data as a sum of log-quantized numbers, or a difference depending on the signs of the constituent numbers, we can achieve arbitrarily precise representations. How many log-quantized numbers to use for such an encoding, which we term *cardinality*, is an important design parameter. Since cardinality has a major impact on encoding efficiency, it must be increased parsimoniously; at the same time, for certain values, required precision can be achieved only by increasing cardinality. Thus it is no coincidence that both proposals [2], [3] stipulate that cardinality is variable, i.e., data-dependent, which seems to be the best way to strike the balance between efficiency and precision. Yet, the data-dependent word length of encoded weights means variable-latency multiplication, posing a significant challenge to designing a scalable hardware architecture.

Also to be relevant for recent deep neural networks (DNNs) such MobileNets [4], which are widely used in the embedded domain, hardware architectures must support efficient mapping of various types of computation including depthwise convolution (DWC), pointwise convolution (PWC), and 3D

convolution. Doing this efficiently when there is a large number of PEs (Processing Elements) performing variable-latency multiplication, is both challenging and necessary to realize the advantages of log quantization on recent DNNs.

In this paper we present a novel hardware architecture for nonuniform multi-word neural networks that addresses the aforementioned challenges. Our proposed architecture is scalable with the number of PEs while maximizing the input/output data reuse, and supports DWC and PWC as well as 3D convolution. To minimize the impact of variable-latency multiplication on performance, we also propose a hardware-software cooperative technique.

Our experimental results using various convolution layers from MobileNetV2 demonstrate the speed advantage of our architecture and high scalability with the number of PEs, compared with previous architectures for depthwise separable convolution or log quantization. Our results also show that our channel reordering technique is very effective in improving the performance of our architecture.

This paper makes the following contributions: (i) a novel hardware architecture with high scalability and data reuse for nonuniform multi-word log-quantized neural networks; (ii) a scheme to support, without refabrication, various types of convolution layers, including DWC, PWC, and 3D convolution; and (iii) static and dynamic channel reordering that can significantly reduce the impact of variable-latency multiplication on performance.

The rest of the paper is organized as follows. We review background and related work in Section II, and present our proposed architecture with examples in Section III. More details of our architecture including schemes to maximize data reuse are given in Section IV. Section V discusses how variable-latency multiplication is handled along with our optimizations. Section VI presents our experimental results, and Section VII concludes the paper.

II. BACKGROUND AND RELATED WORK

A. Linear vs Logarithmic Quantization

Unlike linear quantization, log quantization [1] uses uneven quantization distances, which increase in proportion to the magnitude of value. For an input $0 < |x| \leq \frac{1}{2}$, we define its log quantization, \tilde{x} , as $\text{LogQ}(x) := \text{sign}(x) \cdot \text{round}(-\log_2|x|)$. Conversely $x = \text{sign}(\tilde{x}) \cdot 2^{-|\tilde{x}|}$ ($\tilde{x} = 0$). If $\tilde{x} = 0$, $x = 0$. For a different input range, scaling is applied before quantization.

The logarithmic quantization allows very high precision for small numbers. The smallest positive number that can be represented by R -bit quantization (including sign bit) is $2^{-2^{R-1}+1}$, where R is referred to as *resolution*. On the other hand, the average quantization error can be quite high compared with linear quantization at high resolution. To remedy this problem, recently new techniques have been proposed [2], [3], which is to use multiple words to quantize a single value. For instance, if quantization error $x - \tilde{x}$ is large, another round of log quantization can be invoked, generating an extra word, $\tilde{x} = \text{LogQ}(x - \tilde{x})$. This way, the accuracy of representation is significantly increased with minimal encoding overhead while the hardware advantage of log quantization is retained.

While an arbitrary number of extra words can be used, a limit must be imposed for hardware design. We set the limit to one, which has been shown to be sufficient in practice [3]. A differentiable training algorithm for such networks is also proposed [3].

B. DNN Hardware Accelerators

DNN layers are computationally simple and mostly regular, motivating hardware accelerator approaches [5], [6]. Hardware accelerators for DNNs have been built around an array of MAC (multiply-accumulate) units, which is typically arranged as a 2D or 3D array, to which some loops of the layer computation are mapped. Depending on how it is done, performance (in the number of cycles and clock speed) as well as area and the amount of on-chip memory and off-chip memory accesses all vary. The decision of which loops to map to hardware parallelism is also known as (hardware) *dataflow* [6].

Nonuniform multi-word neural networks, where some weights may have multiple words, pose another challenge to the design of efficient hardware architectures. Having multiple words means that

some MAC operations must take multiple cycles, which leads to a synchronization problem among MAC units or PEs, i.e. wasted cycles. The more weights we use simultaneously, the higher the likelihood of wasted cycle due to weight mismatch. One solution to this problem is to use a dataflow that permits only one weight value in any given cycle; all PEs in the architecture use the same weight value in any cycle, thus no synchronization issue (e.g., [2]). However, such an architecture is hard to scale to a large number of PEs. Minimizing the synchronization overhead while being scalable is an essential challenge of our design problem.

Deep learning is a fast growing field, with increasing numbers of applications and DNNs. But the convolution layer has proven essential and effective in many applications related to image [7], video [8], and even speech [9]. For image-related tasks, depthwise separable convolution (DSC) has shown to be more competitive than 3D convolution when normalized to the number of MAC operations or weight parameters, leading to many state-of-the-art networks such as MobileNets [4], which are very popular in the embedded domain. Depthwise separable convolution consists of a combination of depthwise convolution (DWC) and pointwise convolution (PWC), the latter of which is also called 1x1 convolution. In terms of computational complexity, PWC usually accounts for the majority of workload.

There are previous hardware accelerators for DSC [10]-[12], which we discuss in VI-G.

Finally, our nonuniform multi-word neural network is similar to sparse network in that the number of weight words, and therefore that of unskippable MAC operations, is data-dependent. However, there is an important difference that makes sparse DNN architectures not ideal for our problem. In our network each weight has at least one word and possibly one more, meaning that our architecture must be able to handle efficiently dense computation as well as sparse computation, unlike sparse hardware that only needs to do sparse computation. This explains why our architecture is based on dense processing, handling extra words as exceptions.

III. OUR PROPOSED ARCHITECTURE

Table I: Some symbols used in this paper

Symbol	Description
W_{out}, H_{out}	Width and height of the output feature map
C	Number of input channels
M	Number of filters (output channels)
K, S	Size and stride of 2D convolution filter (in each dim)
T_W, T_H	Width and height of hardware PE plane
N	Number of PE planes

A. Hardware Dataflow

Pointwise convolution accounts for the majority of computation complexity in DNNs employing depthwise separable convolution. For instance, in MobileNetV2 [13] PWC and DWC account for 82.4% and 6.9% of MAC operations, respectively. As such, our primary design target is PWC, but DWC is also supported, since having separate datapaths for DWC and PWC can lead to low resource utilization due to varying ratios of PWC to DWC within and across DNNs.

```

for ( $h = 0$ ;  $h < H_{out}$ ;  $h++$ )
  for ( $w = 0$ ;  $w < W_{out}$ ;  $w++$ )
    for ( $m = 0$ ;  $m < M$ ;  $m++$ )
      for ( $c = 0$ ;  $c < C$ ;  $c++$ )
         $OFM[m, h, w] += W[m, c] \times IFM[c, h, w];$ 

```

Figure 1: PWC kernel.

Figure 1 shows the code for pointwise convolution. IFM and OFM stand for input feature map and output feature map, respectively, and key symbols are defined in Table I. One of the primary issues in architecting a hardware accelerator for such a loop nest is to select the set of iterations (or MAC operations) processed simultaneously by hardware. This often boils down to the problem of deciding which loop level(s) to be mapped to the hardware parallelism, though more elaborate mapping schemes can also be used. For instance, if we intend to use a 2D MAC array (thus the degree of hardware parallelism is two), we need to choose two loop levels that will be mapped to hardware.

Since the PWC kernel reuses the weight parameters along the OFM width and height directions (W_{out} and H_{out} loop), those two loops are the best candidates for hardware parallelism. For higher scalability, if we choose to use a 3D PE array, what should be the next loop level to be mapped to hardware? There are two options, C or M -loop, which are mostly symmetrical.

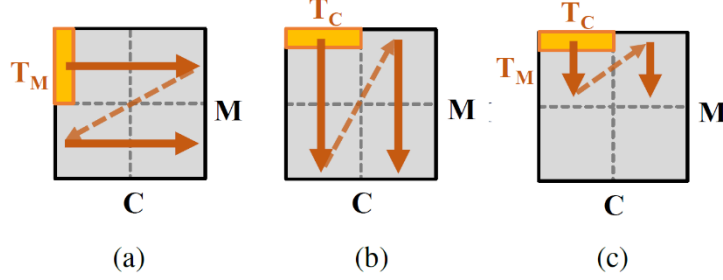


Figure 2: Three ways of parallelizing the M-C loop nest. (a) Output filter parallelism with OFM data reuse (b) Input channel parallelism with IFM data reuse (c) Input channel parallelism with reuse of both IFM & OFM data.

Table II: Mapping options for PWC kernel

The third source of parallelism (Enclosing loop's trip count)		T_M ($\times C$)	T_C ($\times M$)	T_C ($\times T_M$)
IFM regs	Relative size	1	T_C	T_C
	Data reuse factor	1	C	T_M
OFM regs	Relative size	T_M	1	T_M
	Data reuse factor	C	1	C
Datapath similarity (w/ DWC)		Low	High	High

Parallelizing along the M -loop (or C -loop) means that T_M (or T_C) iterations of that loop (called *tile*) are processed simultaneously by the PE array, as illustrated in Figure 2. The figure also shows how the entire weight matrix W of size $M \times C$ can be covered by repeated invocations of the tile, where input or output data reuse can be inferred from the processing order shown by arrows.

Table II summarizes the pros and cons of the two options. In both options, we can exploit data reuse with either IFM or OFM but not both. At first glance, choosing M (i.e., exploiting OFM data reuse) seems better, because OFM is read-and-write whereas IFM is read-only. Yet there is one deterrent: its datapath would be less similar to that of DWC, being devoid of post-MAC adder trees

(see Section III-C). Choosing C instead can lead to a more similar datapath as that of DWC. But then the OFM registers will be live for only one cycle, resulting in constant thrashing of the registers.

Thus we propose the third option combining the two. It is based on the channel parallelism with IFM data reuse (Figure 2b). But to enable OFM data reuse, we increase OFM registers by T_M times

```

for ( $h' = 0$ ;  $h' < H_{out}$ ;  $h' += T_H$ )
  for ( $w' = 0$ ;  $w' < W_{out}$ ;  $w' += T_W$ )
    for ( $m' = 0$ ;  $m' < M$ ;  $m' += T_M$ )
      for ( $c' = 0$ ;  $c' < C$ ;  $c' += T_C$ )
        for ( $m = m'$ ;  $m < T_M$ ;  $m++$ )
          for ( $h = h'$ ;  $h < T_H$ ;  $h++$ )
            for ( $w = w'$ ;  $w < T_W$ ;  $w++$ )
              for ( $c = c'$ ;  $c < T_C$ ;  $c++$ )
                 $OFM[m, h, w] += W[m, c] \times IFM[c, h, w]$ 

```

Figure 4: PWC kernel with tiling.

and tile the M -loop as shown in Figure 3. The fourth innermost loop, whose trip count is T_M , is

```

for ( $h = 0$ ;  $h < H_{out}$ ;  $h++$ )
  for ( $w = 0$ ;  $w < W_{out}$ ;  $w++$ )
    for ( $c = 0$ ;  $c < C$ ;  $c++$ )
      for ( $k_h = 0$ ;  $k_h < K$ ;  $k_h++$ )
        for ( $k_w = 0$ ;  $k_w < K$ ;  $k_w++$ )
           $OFM[m, h, w] += W[m, c] \times IFM[c, h, w]$ 

```

Figure 3: DWC kernel (for $S=1$)

executed iteratively, but there are enough OFM registers for the T_M iterations, that OFM registers will be fully reused instead of being thrashed. This option requires T_C times as many input registers as choosing T_M (i.e., output filter parallelism), and incurs corresponding load bandwidth increase for IFM registers. However, by making $T_M = T_C$ we can ensure that the *average* bandwidth does not increase over that of choosing T_M .

B. Consideration for Other Types of Convolution

While 3D convolution can be easily implemented as either DWC or PWC, DWC has a very different computation pattern from PWC, and needs a special consideration.

Figure 4 shows the code for DWC. Like other types of convolution, DWC shares weight parameters along the OFM width and height directions (W_{out} , H_{out}), which are the obvious choice for our

hardware parallelism. This leaves us the question of what the third loop should be. Since the two K loops have usually very few trip counts, we can flatten them into one, hence called K^2 loop. After this arrangement, we have only two options: the flattened K^2 or C loop.

If we choose the C loop, the DWC computation is completely independent among the iterations of the C loop, which means that there will be few wasted cycles due to synchronization. Thus this scheme may offer higher performance, but it requires rather large IFM/OFM registers due to the lack of data reuse along the C loop.

On the other hand, if we parallelize along the K^2 loop, we can exploit data reuse with both (i) OFM registers along the K^2 loop and (ii) IFM registers within the loop nest involving the W_{out} , H_{out} , and K^2 loops. The IFM reuse is evident from the fact that the number of IFM elements needed for the loop nest is less than the total number of iterations. This means that this scheme can have an advantage in terms of input/output data transfer (e.g., fewer input/output registers). Moreover, we can mitigate the synchronization overhead of the second option by a scheduling trick (Section V-A). Thus we choose the K^2 loop as the third source of parallelism.

Table III: Hardware dataflow summary

Kernel	HW-mapped loops	Latency w/ single-cycle mult.
PWC	H_{out}, W_{out}, C	$\left\lceil \frac{C}{N} \right\rceil M \cdot \left\lceil \frac{W_{out}}{T_W} \right\rceil \left\lceil \frac{H_{out}}{T_H} \right\rceil$
DWC	H_{out}, W_{out}, K^2	$\left\lceil \frac{K^2}{N} \right\rceil \left\lceil \frac{W_{out}}{T_W} \right\rceil \left\lceil \frac{H_{out}}{T_H} \right\rceil \cdot C$
3D Conv	H_{out}, W_{out}, C	$K^2 \cdot \left\lceil \frac{C}{N} \right\rceil M \cdot \left\lceil \frac{W_{out}}{T_W} \right\rceil \left\lceil \frac{H_{out}}{T_H} \right\rceil$
	H_{out}, W_{out}, K^2	$\left\lceil \frac{K^2}{N} \right\rceil \left\lceil \frac{W_{out}}{T_W} \right\rceil \left\lceil \frac{H_{out}}{T_H} \right\rceil \cdot C \cdot M$

Table III summarizes hardware dataflows for different convolution kernels, which are all output stationary. The pressure to minimize synchronization overhead due to mismatched weights has resulted in similar dataflows that stretch image width and height, but the third source of parallelism varies.

C. Main Datapath

Based on the discussion in the previous section, we propose a common accelerator architecture to efficiently support different types of convolution while being robust to multiplication latency that varies with weight data.

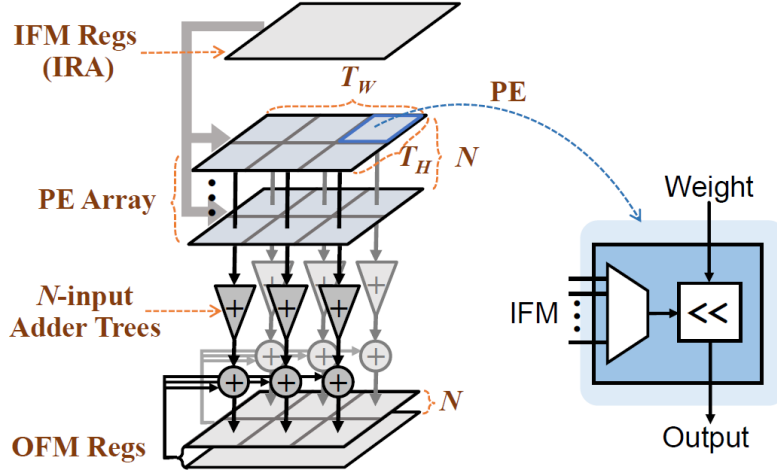


Figure 5: The datapath of our proposed architecture.

Motivated by the three operations needed to realize our PWC dataflow as illustrated in figure 2c, which are pointwise multiplication, sum across channels, and output channel-wise accumulation, we design our datapath to be a sequence of 3D array of PEs (Processing Elements), adder trees, and accumulating adders, as illustrated in Figure 5. Each PE consists of just a multiplier or its equivalent (e.g., shifter for log quantization) and a mux; the mux is for DWC only, and adders are conspicuously missing from PEs. By removing adders, which are usually included in PEs, our PEs can be slimmer; yet, both DWC and PWC dataflows of ours can be successfully mapped to the datapath, which we show in the next section.

It is convenient to consider the 3D PE array as a stack of *PE planes*, where each PE plane is a $T_H \times T_W$ array of PEs. A PE plane naturally maps to a tile in the first two parallelized loops in Table III, which are the same for all three types of convolution; the third parallelized loop is mapped to the depth of the stack, which is denoted by N . Since a PE plane is mapped to a tile in the $H_{out}-W_{out}$ loop nest (see Figure 3, Figure 4), each PE plane needs only one weight parameter at any given cycle, regardless of the type of convolution. In other words, all PEs in a PE planes are always in sync.

There are $T_H \times T_W$ adder trees, each of which takes N multiplication results from the corresponding PEs in the PE array, and produces one number, which is the sum of the N inputs. The output of adder trees goes through accumulating adders, and saved in the output registers.

For PWC, both IFM and OFM registers need to be arranged as a stack of *register planes*, which has the same dimension as the PE array. But for DWC it is convenient to view the IFM registers as a 2D array, which we call *Input Register Array* (IRA). We discuss IRA sizing in Section IV-C.

D. Main Examples

We now illustrate how the main datapath works. Here we assume that multiplication takes one cycle; extension to multi-cycle multiplication is presented in Section V.

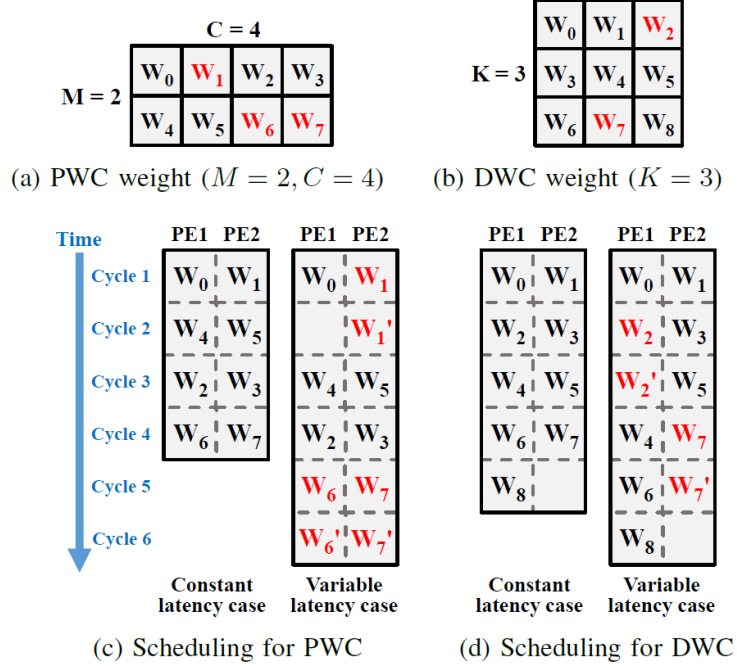


Figure 6: Scheduling for variable latency

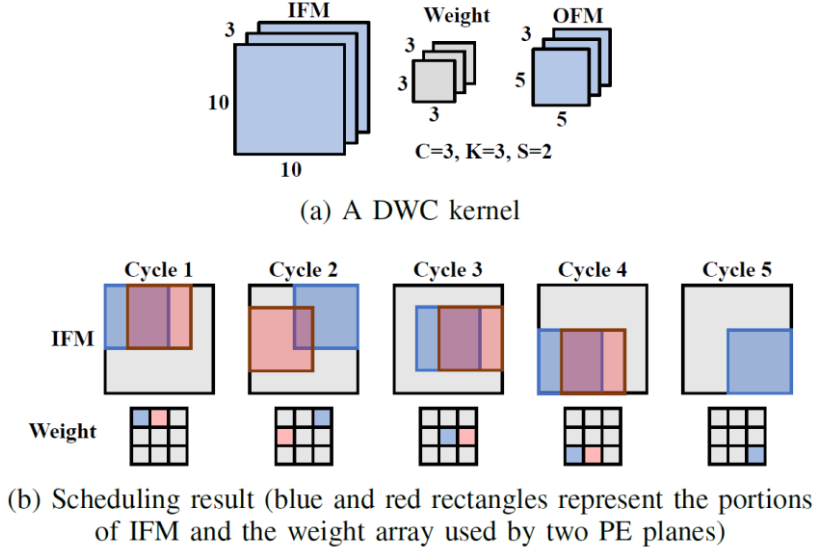


Figure 7: DWC mapping example.

1) PWC: Since the PWC computation is the same across points, we only need to consider one point, which corresponds to one PE per PE plane, or N PEs for the entire PE array. Note that N is equal to $T_C (= T_M)$ in Figure 2c.

Because we parallelize along the C -loop, the output from N PEs needs to be summed, which can be obviously done by adder trees. What is not very obvious is how to support the zig-zag processing order in Figure 2c and make sure that the result of adder trees are accumulated into the right set of OFM registers. Consider the example weight matrix in Figure 6a. Let $N = 2$.

Figure 6a shows the scheduling of the weight elements on two PEs. Note again that these PEs are representative of PE planes, as all PEs in the same PE plane use the same weight. Due to the zig-zag processing order, the final OFM data will be produced at cycles 3 and 4 for our example. If $M = 3$, the next OFM data will be finalized at cycle 6. In general, generating OFM data of one PE plane size for all input/output channels takes $\lceil C/N \rceil \cdot M$ cycles, which is to be repeated for the entire width/height of OFM. Managing output data is more cumbersome (explained in detail in Section IV-A). The basic idea is to have $N (= T_M)$ sets of output registers, to one of which we accumulate the result of adder trees.

2) DWC: To illustrate how DWC can be mapped to our architecture, we use the example in Figure 7a. For this example we assume $T_W = 4, T_H = 4, N = 2$, and the IRA size is 9×9 (see IV-C for IRA sizing). In our architecture all PE planes work simultaneously on the same channel, dividing the K^2 iterations among N PE planes. For our example kernel, Figure 7b illustrates how 2 PE planes

perform DWC convolution for one channel in 5 cycles ($N = \lceil 3^2/2 \rceil$), assuming multiplication takes one cycle.

Being output stationary, the output of the PE array during the 5 cycles, which is $T_W \cdot T_H \cdot N$ multiplication results per cycle, corresponds to the same set of $T_W \times T_H$ OFM pixels. Thus we only need to accumulate them both spatially (across stack depth N) and temporally (across $\lceil K^2/N \rceil$ cycles), which is done by adder trees and accumulators, respectively. DWC uses only $T_W \times T_H$ OFM registers. On the other hand, providing input is more complicated, which is explained in detail in Section IV-C.

So far we have generated OFM data of one PE plane size ($T_W \times T_H$) for one channel. To complete the entire kernel, we first finish the entire OFM for one channel, then move to the next channel. This way, weight parameters and IFM data can be better reused than the alternative (first channel-loop then OFM-loop).

Table III shows the number of cycles for each convolution type assuming single-cycle multiplication operations.

IV. MAXIMIZING INPUT AND OUTPUT REUSE

A. Architecture Overview and OFM Data Path

At the top level, our proposed architecture has the main datapath including IFM, weight, and OFM registers and three main on-chip buffers (IFM, weight, and OFM) and one very small buffer (index), which are all double-buffered to overlap DMA latency with computation time (see Figure 8).

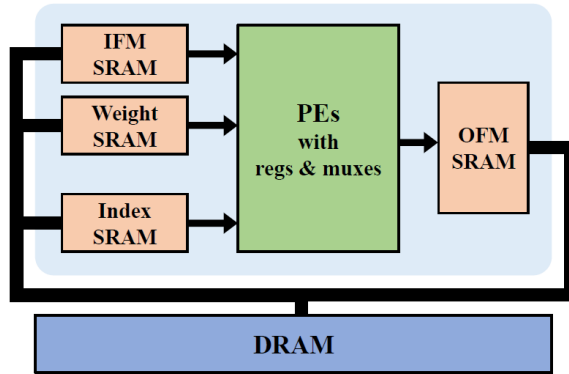


Figure 8: Overall architecture of our proposed accelerator.

Being output stationary, our architecture has a relatively simple OFM data path. The OFM registers form a stack of *register planes*, with the whole stack having the same dimension as the PE array, i.e., $T_W \times T_H \times N$. For PWC, all register planes are used in a round-robin fashion, which is crucial to fully reusing OFM register data. Specifically, the result of adder trees is accumulated to one of the register planes. This is achieved by employing only $T_W \times T_H$ accumulating adders, but each accumulating adder is attached to an N -to-1 mux and a 1-to- N demux to redirect its input and output from/to the right register plane, which is cheaper than replicating accumulating adders themselves by N times. Once final OFM data is generated, it is copied to the on-chip OFM buffer.

B. Weight Data Path

The weight data path is very simple, as the architecture requires only N weight parameters in any given cycle, which can be provided by the on-chip weight buffer via N weight registers. The weight buffer is also N -word wide. While SLQ uses extra words for some weight values, it does not complicate our accelerator at all, since the exact locations of the extra words can be determined in advance and stored in the main memory in the order they are used by the hardware. For this to work, the number of PE planes also needs to be known.

Take Figure 6d for instance (the variable latency case). The weight parameters are realigned in N -word bundles already in the main memory, which are copied to the weight buffer, and used by the PE array. The PE array knows that W'_2 and W'_7 are extra words, and therefore can get the cycle information correct (which is needed to control in-PE muxes; see next section).

C. IFM Data Path

IFM registers are organized in a 2D array of registers, called IFM register array (IRA). Unlike some previous work, the IRA consists of isolated registers; that is, registers in IRA are not connected to each other. With no mechanism to move data around inside IRA, PEs can still access IFM in different ways (which is crucial for DWC and switching between DWC and PWC) through in-PE muxes, the size of which is determined by the number of different ways of accessing IFM. We next discuss the requirements on the sizes of IRA and in-PE muxes.

1) *DWC*: For DWC, IRA size should be no less than $(K + S(T_H - 1), K + S(T_W - 1))$. Once loaded, the IRA is used by the PE array for at least T consecutive cycles, where $T = \lceil K^2/N \rceil$, which means that the in-PE muxes should be T -to-1, if the DWC kernel is the only kernel supported. Note that the mux size requirement does not increase if supporting two or more DWC layers that have the same set of K and S parameters.

2) *PWC*: PWC requires IRA to have at least $T_H T_W N$ words. In this case, PEs access IRA in only one way; in other words, PEs are one-to-one mapped to IRA data. Since the IRA size requirements are different between DWC and PWC, the final IRA size is given by the greater of the two¹. The final mux size is given by the sum.

¹ The aspect ratio does not really matter because it is a register array, but from the perspective of coding, it is more convenient if IRA size covers DWC size requirement(s).

3) *Example*: Table IV lists the various types of layers used in MobileNet V2 as well as the required IRA and mux sizes.

Table IV: Determining IRA size for MobileNet V2
 $(T_W, T_H, N) = (8, 8, 4)$

Layer type	IRA size (word)	T	Mux size
DWC ($S = 1, K = 3$)	10×10 (= 100)	3	3
DWC ($S = 2, K = 3$)	17×17 (= 289)	3	3
PWC	$8 \cdot 8 \cdot 4$ (= 256)	4	1
On the network level	17×17	-	7

4) *IFM Buffer*: If IRA is large, it may be difficult to design a very wide SRAM buffer to load all IFM registers at once. Luckily, since IFM registers are reused over T cycles, we can load IFM registers over T cycles using $T \times$ narrower IFM buffer. For instance, in the MobileNet V2 example the DWC layer with $S = 2$ requires 289 words. Assuming the IFM buffer is 128-byte wide and a word is 8-bit, loading IFM registers can be done in 3 cycles ($= \lceil 289/128 \rceil$), which is no greater than the computation latency and therefore can be completely hidden. To hide register load latency, shadow latches can be used.

V. HANDLING VARIABLE LATENCY OPERATIONS

SLQ represents some weight values using two words. While extra words are crucial to achieving high accuracy with log quantization, the same extra words incur not only extra cycles but also wasted cycles due to synchronization. To minimize the latter is our design goal. Since a PE plane uses a single weight value, we need to consider only across multiple PE planes. Thus in this section we consider the problem of scheduling multiple PEs, each belonging to a different PE plane, in the presence of extra weight words.

A. DWC

Consider a DWC weight matrix in Figure 6b, where two weight values shown in red need extra words in SLQ. Figure 6d shows how the order of weight processing is changed due to extra words, where the number of PE planes is assumed to be 2 ($N = 2$). Note that W'_2 must be processed by the same PE plane as that which processes W_2 , since each weight value requires connections to a specific set of IFM registers, and those connections are made specifically for each PE plane via in-PE muxes. In other words, only W_2 uses the exact same set of IFM registers as W'_2 does, which creates a scheduling constraint. Note also that when PE1 is delayed by one cycle due to an extra weight W'_2 , PE2 can proceed unaffected, which is contrary to PWC. This is because in DWC each PE plane maintains its own cycle counter and controls in-PE muxes independently of other PE planes, and also because all the PE planes share the same input and output registers. Thus in DWC extra words create wasted cycles only due to the imbalance among PE planes in the *total* number of extra words for each channel, which is relatively small.

B. PWC and Column Reordering

Recall that the $M \times C$ weight matrix of PWC is processed in units of $1 \times T_C$ submatrices, which we now call *bundles*, and that the number of bundles determines the latency of PWC. A bundle containing even a single extra weight means an extra cycle for the entire bundle; i.e., the other PE planes cannot proceed with the next cycle's work. This is because every cycle in PWC is for either different input or different output.

On the other hand, we can reorder the columns of the weight matrix such that extra weights happen together in the same bundle or none at all. This can be done statically, by changing the (input) channel

ordering of the current layer *and* the (output) filter ordering of the previous layer simultaneously, which does not require any hardware or runtime overhead at all. Or it can be done dynamically at the granularity of $T_M (= N)$ rows, by manipulating the column addresses when reading from the on-chip IFM buffer to IFM registers, in addition to offline reordering of weights.

For dynamic column reordering, we need a small on-chip index memory (called *index buffer*) that can give, each cycle, the base address or index of the next N columns' corresponding IFM data. Thus the width needs to be N words, and the amount of indices for the entire IFM buffer is very small ($T_H T_W$ times smaller than the IFM buffer size). Though we need a different set of indices for every T_M rows, it can be supplied from the off-chip memory at regular interval via the index buffer.

C. Column Reordering Algorithm

Algorithm 1 Fast Column reordering

Input: A set of bit-vectors and a parameter N

Output: Initially every bit-vector is a group by itself.

Step 0: Initially every bit-vector is a group by itself.

Step 1: Create an interference graph, where nodes are groups and the weight of an edge (v_1, v_2) is the number of wasted slots of merging v_1 and v_2 to a new group.

Step 2: Remove a least-weighted edge along with the nodes connected to it. The removed nodes form a new group passed to the next round. Repeat this until all nodes are removed.

Step 3: Go to Step 1 unless group size is N or the largest 2^k no greater than N .

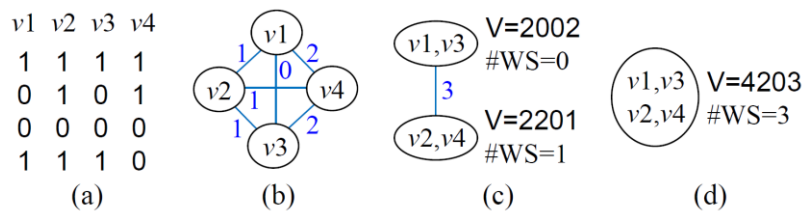


Figure 9: Column grouping example (Algorithm 1).

The problem of finding the best column ordering can be formulated as an optimization problem: Given an integer N and an $M \times C$ binary matrix, find an ordering of matrix columns that minimizes the number of size- N , aligned bundles containing at least one. The binary matrix is constructed such that an element is 1 if the corresponding weight value requires extra word, otherwise 0.

This problem can be viewed as partitioning C bit-vectors into $\alpha = \lceil C/N \rceil$ groups whose size is at most N , which reveals that the number of unique orderings is $C!/((N!)^\alpha \cdot \alpha!)$, assuming N divides C . Also a special case of the problem with $\alpha = 2$ is the graph partition problem, which is NP-hard; hence, we propose a heuristic algorithm.

First we observe that the optimal solution minimizes the number of *wasted slots*. For instance, for the weight matrix of Figure 9a, if $N = 4$, the second bundle, 0101, takes two cycles, wasting two slots, but the first or third bundle wastes no slot. We can extend this idea to entire columns. The number of wasted slots when grouping v_1 and v_2 together is 1, and so on. Then we would like to partition columns such that each partition (i.e., group of columns) has the fewest wasted slots.

The problem is easier if $N = 2$, in which case we can pair most similar bit-vectors, which we do greedily in Step 2 of the algorithm (see Figure 9b). The challenge is how to quickly estimate the number of wasted slots of merging two groups, which is necessary to compute interference (or dissimilarity) in Step 1. This can be done accurately and efficiently. First, each group is associated with an integer vector V (called bitcount), which is the bit-vector itself if the group size is 1, or $V + U$ if merging two groups with bitcounts V and U . Then, the number of wasted slots of a group with bitcount V is given as $\sum_i (n - V_i) \% n$, where n is the size of the group and $\%$ is the modulo operation. Since Step 1 can be done in $O(C^2)$, the overall complexity of Algorithm 1 is $O(C^2 \log N)$.

If N is not a power of 2, the last few rounds proceed as follows. First, select the top α groups with fewest wasted slots. Second, dismantle all remaining groups into bit-vectors. Third, compute the interference between every pair of a group and a bit-vector, forming a bipartite, interference graph. Fourth, find an edge with the least weight and remove it along with the nodes connected, creating new groups for the next round, which is repeated until all group nodes are removed. Steps 3 and 4 are repeated until there is no remaining bit-vector node.

Some bit-vectors may be identical from the beginning. Thus we find all identical bit-vectors in the first-round interference graph, where zero edge weight means that the two bit-vectors are identical. Since N identical bit-vectors immediately form a completed group, we need only consider how to handle fewer than N identical bit-vectors: we split them into groups of 2^k bit-vectors ($k \geq 0$) and add each 2^k -sized group to the appropriate rounds. Note that our algorithm can be used for both dynamic and static reordering schemes.

VI. EXPERIMENTS

A. Experiment Setup

To evaluate the efficiency of our proposed architecture we use MobileNetV2 [13], which shows nearly the state-of-the-art performance (per MAC operation) for ImageNet classification. Our technique is applied to all PWC and DWC layers. We have retrained the network with Caffe [14] using log quantization [2] starting from pretrained weights. After confirming sufficient test accuracy, the trained weights are used to evaluate the performance of our architecture.

The architectural parameters are as shown in Table IV including the IRA and mux size, making the total number of PEs 256, for easy comparison with the DaDianNao architecture [5], which has the same number of PEs. We have implemented our architecture (and the MVM case for comparison) in Verilog HDL, and synthesized it with Synopsys Design Compiler using Samsung 65~nm library. Results for on-chip memories are estimated with Cacti 7.0 [15].

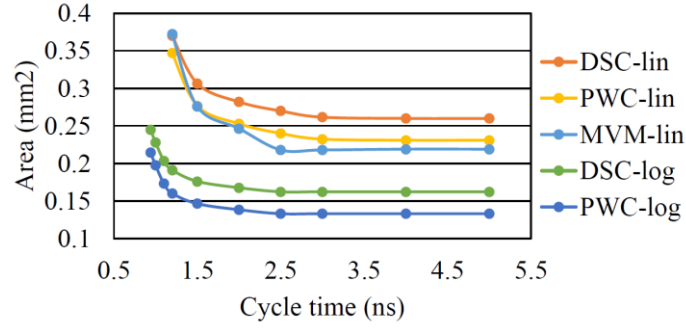
For performance evaluation we have developed a DNN simulator extending Caffe [14] that not only verifies the correct functionality of the DNN but also generates the accurate cycle count, taking into account the architectural parameters such as PE array size and weight values.

B. Hardware Synthesis Result

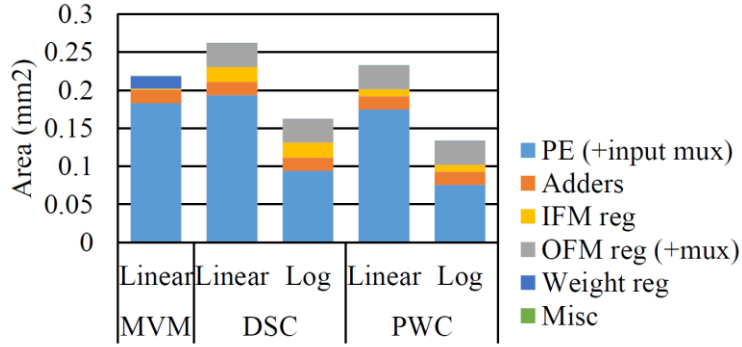
In this section we compare the following cases:

- 1) **MVM**, which serves as the baseline, represents the 2D MAC array structure employed by DaDianNao [5], which is chosen due to its high visibility and straightforward design. This is also representative of [12] designed for MobileNetV2.
- 2) **DSC** is our proposed architecture supporting both DWC and PWC.
- 3) **PWC** is our proposed architecture supporting PWC only (input muxes removed, fewer IFM registers), which may be useful for accelerating PWC layers only or for 3D convolution-based networks (e.g., AlexNet and SqueezeNet).

In addition, to make it easy to see the differences we show the results of the linear-quantized versions of DSC and PWC even though they are *not* the main comparison. We use 10-bit input activation and 4-bit log-quantized weight or 9-bit linear-quantized weight (all including sign bit), which show near-baseline performance in our training of MobileNetV2 (see Section VI-E).



(a) Area vs. delay



(b) Area breakdown ($T_{cyc} = 3$ ns)

Figure 10: Hardware synthesis result.

Figure 10a shows the area and delay of the various architectures. The lower ones are the log-quantized designs and the upper ones the linear-quantized ones, which clearly shows the area/delay advantage of the proposed log-quantized architecture. An iso-area comparison at about 0.22~mm^2 suggests that our log-quantized hardware is about $2.5\times$ faster than the MVM case, demonstrating the strength of our architecture for applications where high speed is required. The actual performance of our architecture will be lower due to extra words in weight parameters, the overhead of which comes to about 50% additional cycles (see the next section). On the other hand, the MVM case also is not quite ready for MobileNets, as depthwise separable convolution requires more reconfigurable architectures such as ours and [10], with corresponding area overhead.

Figure 10b shows the area breakdown for $T_{cyc} = 3\text{ns}$. As expected, the linear versions of DSC and PWC show increased area compared with the MVM baseline, mostly due to IFM/OFM registers. But the PE area reduction in log-quantized versions more than compensates for the increase. The IFM registers take less area than OFM registers, which is because the OFM register area includes that of output muxes, and the PWC architecture uses fewer IFM registers, which can be implemented as

latches. In Figure 10a one can observe that when the cycle time is very small ($T_{cyc} = 1.2\text{ns}$), the MVM case has the largest area, even more than the other linear cases. This area increase is due to the adder trees, which being larger than those of our architecture (16-input vs 4-input), need much larger circuitry to meet the timing requirement.

C. Performance and Effect of Column Reordering

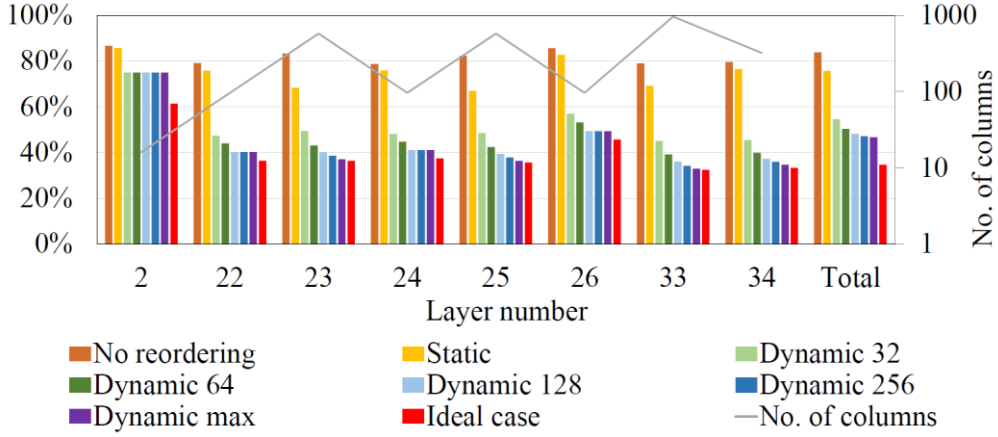


Figure 11: Cycle overhead due to extra words in PWC layers.

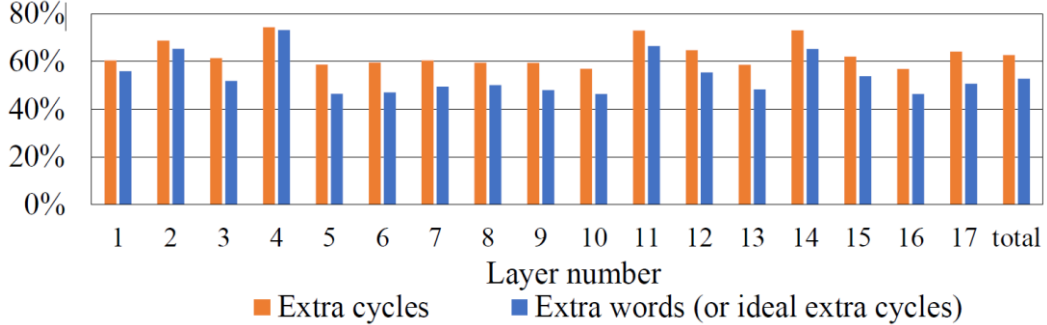


Figure 12: Cycle overhead due to extra words in DWC layers.

Figure 11 shows the performance overhead of using SLQ, showing only the highest impacting layers (in latency) out of all 34 PWC layers of MobileNetV2. Unlike the CNNs used in the previous work [2], MobileNetV2 turns out to use much more extra words for encoding its weights: 35 extra words per 100 weight parameters for PWC layers on average (see the *Ideal case* in the graph). Ideally, these extra words consume just 35% more cycles, which is the case if $N = 1$, but require far more cycles if N is greater. The graph shows that extra cycles amount to 84% of the *base latency* if column reordering is not used, where *base latency* is that of using single-cycle multipliers. The static column

reordering has a very limited success, which is due to the large column length; when columns are long, interference among columns tends to be high.

Dynamic column reordering, on the other hand, is very effective in reducing the overhead of extra words, often minimizing it to nearly the ideal level. Unlike static, the dynamic scheme has a limited scope, which we vary from 32 columns to unlimited. (Note that even in the dynamic scheme, the reordering information itself is generated statically before runtime, and at runtime it is only used.) Quite understandably, increasing the scope generates higher performance; when it doesn't, it is often because there are not enough columns in the weight matrix. Layer 2 is a good example: while it has an exceptionally high overhead, it is because the weight matrix has only 16 columns.

The weights of DWC layers, Figure 12 suggests, tend to require significantly more extra words than those of PWC layers, which may be due to their smaller number. Fortunately, however, the performance overhead due to extra words in DWC is only marginally higher than the extra words, thanks to our scheduling (see Section V-A).

D. Effect of the Number of PE Planes

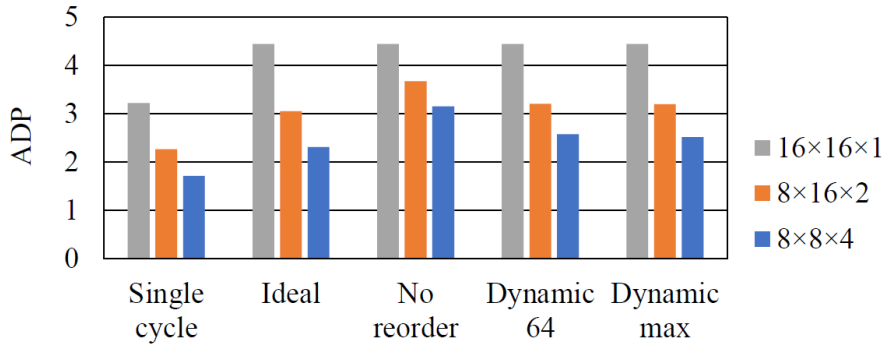


Figure 13: ADP vs Number of PE planes (PWC layers only).

A distinguishing feature of our proposed architecture is the number of PE planes, N . To evaluate the importance of having this dimension, we vary N from 1 to 4 while holding the total number of PEs constant. Note that the single-plane architecture ($N = 1$) corresponds to previous work [2], though it supports 3D convolution only. For this comparison we use all PWC layers of MobileNetV2, with the same settings as before. Figure 13 compares the area-delay product (ADP) of the three architectures, under different column reordering schemes (T_{cyc} is 3ns). Because the three architectures have very similar areas, nearly all the ADP difference comes from the latency.

From the result we glean the following facts. First, we note that the base latency (i.e., that of using single-cycle multipliers) works favorably to the 4-plane architecture, which is the advantage of 3D arrays over 2D arrays especially in layers with shrinking height and width as is the case with the last layers of representation learning DNNs. Second, the performance overhead due to multiple PE planes, which ultimately comes from extra weight words, does not negate the advantage. Note that the amount of extra words is the same regardless of N (i.e., the *Ideal-to-Single-cycle* ratio in the graph is constant), but the *conversion rate* from extra words to extra cycles is different. The graph shows that the performance advantage of multiple PE planes becomes quite offset if no column reordering is used, but is mostly recovered by dynamic reordering. This result suggests that the additional dimension proposed by our architecture can indeed make a significant improvement to performance. Also dynamic column reordering is essential to realizing the improvement.

E. Training Result

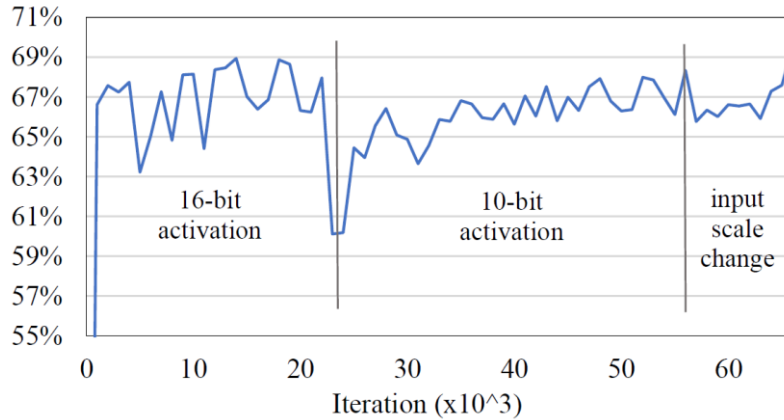


Figure 14: Top-1 accuracy during retraining of MobileNet V2

Figure 14 shows top-1 validation accuracy during our training of MobileNetV2 with SLQ quantization. One epoch corresponds to 5000 iterations. Initially we use 16-bit activation and 4-bit SLQ weight for all layers, starting from a full-precision pretrained model, which gives around 71% top-1 accuracy for ImageNet. After reaching 69% accuracy, which is our target, we reduced the activation precision to 10-bit. Finally, after adjusting input activation scales, the accuracy reached 69% with 4-bit SLQ. Throughout our training, the ratio of extra words for weights has remained largely constant.

F. Scalability

To evaluate the scalability of our architecture, we compare the performance, in the number of cycles, of various architectures, including previous MobileNet-targeting architectures **TCAS-II** [10] and **MVM** [12] and a SLQ-based architecture, **DAC19** [2]. Here we use the dataflow information only, and map MobileNetV2 layers (both DWC and PWC) to each dataflow, while varying the number PEs. This essentially measures the PE utilization. We assume single-cycle multipliers for SLQ-based architectures (ours and DAC19).

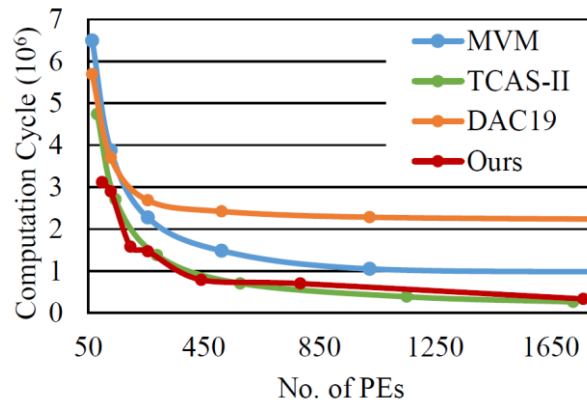


Figure 15: Computation latency vs Number of Pes.

Figure 15 shows the result, which shows that 2D dataflows (MVM and DAC19) have worse scalability than 3D (ours) or 4D (TCAS-II) dataflows, which is quite understandable. Moreover, the performance of ours (as measured in simple latency) is competitive with TCAS-II, though the latter shows better performance and scalability when the number of PEs is over 500.

G. Comparison with Previous Work

Recently hardware architectures targeting MobileNetV2 have been proposed [10]-[12], which all take different approaches. Wu et al. [11] proposes an architecture with separate datapaths for DWC and PWC, which means high efficiency at the layer level but possible low efficiency at the network level, due to pipeline imbalance between DWC and PWC “stages”. Since MobileNetV2’s MAC-operation-count ratio between DWC and PWC varies quite a lot across the network, there is a significant limit to optimizing the throughput of DWC and PWC statically. Also one optimized architecture may become easily suboptimal for another DSC-based network. Liu et al. [12] proposes an architecture based on a MVM (Matrix-Vector Multiplication) structure, which is a 2D MAC array

with its two dimensions mapped to C and M -loops. While this dataflow maps well to PWC, there is no efficient way to map DWC to it. Their solution is to size the dimensions such that the un-utilized dimension, since it is not possible to utilize both dimensions for DWC, has small length. Lastly, Bai et al [10] proposes an architecture that are most similar to ours, in that its datapath can support both DWC and PWC. Such reconfigurability comes with hardware overhead, but their solution gives high PE utilization. Coincidentally all the above architecture was evaluated on FPGAs (and different FPGAs), with no information about area or clock frequency of any standard cell implementation, which makes straightforward comparisons between them or with our architecture difficult.

The main advantage of our architecture compared with the previous DSC architectures is that ours is especially fast, being based on shifters instead of multipliers (see Figure 10a). Second, in terms of area, ours is slightly larger than the simplest one [12], but [10], which is the most elaborate one, is also more complex, and should be larger, than [12]. But area depends on clock speed, and again for high speed applications, our datapath can actually be smaller than even the simplest one, as shown by our experiment. Third, one disadvantage of our architecture is its variable cycle latency, and additional cycle overhead, which is about 50% for PWC and low compared with the area-delay advantage of our architecture. Fourth, our dataflow is very scalable with the number of PEs, especially compared with all 2D array based ones, which includes [2], [12], and on a similar level with [10].

VII. CONCLUSION

We presented an architecture for nonuniform multi-word neural networks, showing its advantages and scalability over previous state-of-the-art architectures. While the primary application of our architecture is log-quantized networks, it could be applied to any neural network if multiplication has variable latency (e.g., [16]). We plan to apply and evaluate our architecture to more diverse CNNs, and also develop a methodology to optimize the architectural parameters, which is not straightforward due to complex dependence between performance, weight values, and architectural parameters (e.g., the number of PE planes).

REFERENCES

1. E. H. Lee et al., “LogNet: Energy-efficient neural networks using logarithmic computation,” in 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2017.
2. S. Lee et al., “Successive log quantization for cost-efficient neural networks using stochastic computing,” in 2019 56th ACM/IEEE Design Automation Conference (DAC), 2019, pp. 1–6.
3. R. Ding et al., “Flightnns: Lightweight quantized deep neural networks for fast and accurate inference,” in 2019 56th ACM/IEEE Design Automation Conference (DAC), 2019, pp. 1–6.
4. A. G. Howard et al., “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” 2017.
5. Y. Chen et al., “Dadiannao: A machine-learning supercomputer,” in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014, pp. 609–622.
6. “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” IEEE Journal of Solid-State Circuits, vol. 52, no. 1, pp. 127–138, 2017.
7. A. Krizhevsky et al., “ImageNet classification with deep convolutional neural networks,” in Advances in Neural Information Processing Systems 25, F. Pereira et al., Eds. Curran Associates, Inc., 2012, pp. 1097–1105.
8. J. Redmon et al., “You only look once: Unified, real-time object detection,” in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779–788.
9. O. Abdel-Hamid et al., “Convolutional neural networks for speech recognition,” IEEE/ACM Transactions on Audio, Speech, and Language Processing, vol. 22, no. 10, pp. 1533–1545, 2014.
10. L. Bai et al., “A cnn accelerator on fpga using depthwise separable convolution,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 65, no. 10, pp. 1415–1419, Oct 2018.
11. D. Wu et al., “A high-performance cnn processor based on fpga for mobilenets,” in 2019 29th International Conference on Field Programmable Logic and Applications (FPL), Sep. 2019, pp. 136–143.
12. B. Liu et al., “An fpga-based cnn accelerator integrating depthwise separable convolution,” Electronics, vol. 8, p. 281, 03 2019.
13. M. Sandler et al., “Mobilenetv2: Inverted residuals and linear bottlenecks,” in 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 4510–4520.
14. Y. Jia et al., “Caffe: Convolutional architecture for fast feature embedding,” arXiv preprint arXiv:1408.5093, 2014.
15. R. Balasubramonian et al., “Cacti 7: New tools for interconnect exploration in innovative off-chip memories,” ACM Transactions on Architecture and Code Optimization, vol. 14, pp. 1–25, 06 2017.
16. H. Sim et al., “A new stochastic computing multiplier with application to deep convolutional neural networks,” in 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), June 2017, pp. 1–6.